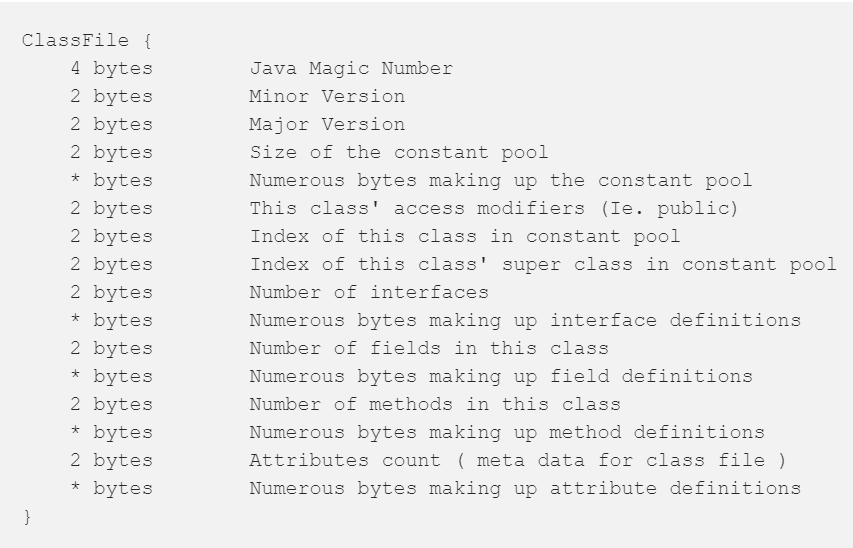
**Writing Hello World in Java byte code**

Taken from <https://medium.com/@davethomas_9528/writing-hello-world-in-java-byte-code-34f75428e0ad> and modified by Luke Lambert.

* First I would suggest opening the above link, as copy/paste will be very useful here
* Don’t worry, we’ll use hexadecimal instead of binary
* When you write a Java program and compile it, the result is a **class file**
  + This class file is Java byte code – a binary data file that contains instructions for the java Virtual machine to execute your program
* Let’s examine the structure of a class file:



* Let’s break down that Class File definition a bit

**Java Magic Number**

* It is four bytes that are always at the start of your file
  + Indicates that your file is a Java class file
* The four bytes are: CA FE BA BE or “Café Babe”

**Version**

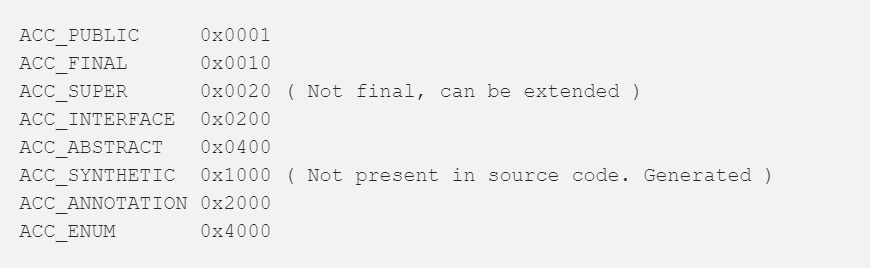
* The next 4 bytes are two 2 byte constructs that make up the version
  + Java 8 is major version 52.0 or 00 00 00 34 in hexadecimal

**Constant Pool**

* The next two byte are highly important; they indicate the size of the constant pool
* The constant pool is the longest and most important part of our program
* Class files contain a lot of UTF8 character data, along with typing information for the character data.
  + Ex. your main method, your class name, and references to other classes
* Everything your class file uses will be here, and surprisingly even for a simple Hello World program there is a bit
* Other areas of the byte code like method definitions will reference into the constant pool table via indexes
  + The constant pool starts at index 1 and goes until size – 1
* After the 2 bytes for the constant pool size, comes the constant pool
  + This is of variable byte length and is dependent on the data contained within
  + Each entry starts with a tag, which tells us how many bytes long that entry will be

**Access Modifiers**

* After the constant pool completes, we have the access modifiers. This is based on a combination of the following:



* We get the access modifiers for the specific class we are defining.
  + For our Hello World program, we can just use 0021 – Super Public

**Class Constant Pool References**

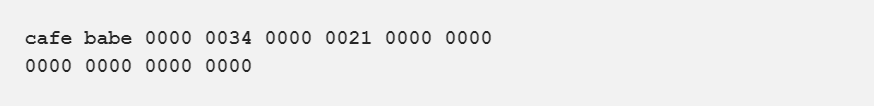
* The next 4 bytes are the class indexes in the constant pool.
  + 2 bytes represent a reference to this class
  + 2 bytes represent a reference to its super class
    - All classes have a super class, even if you don’t declare one, in which case it is **java/lang/Object**

**Interfaces, Fields, Methods, and Attributes**

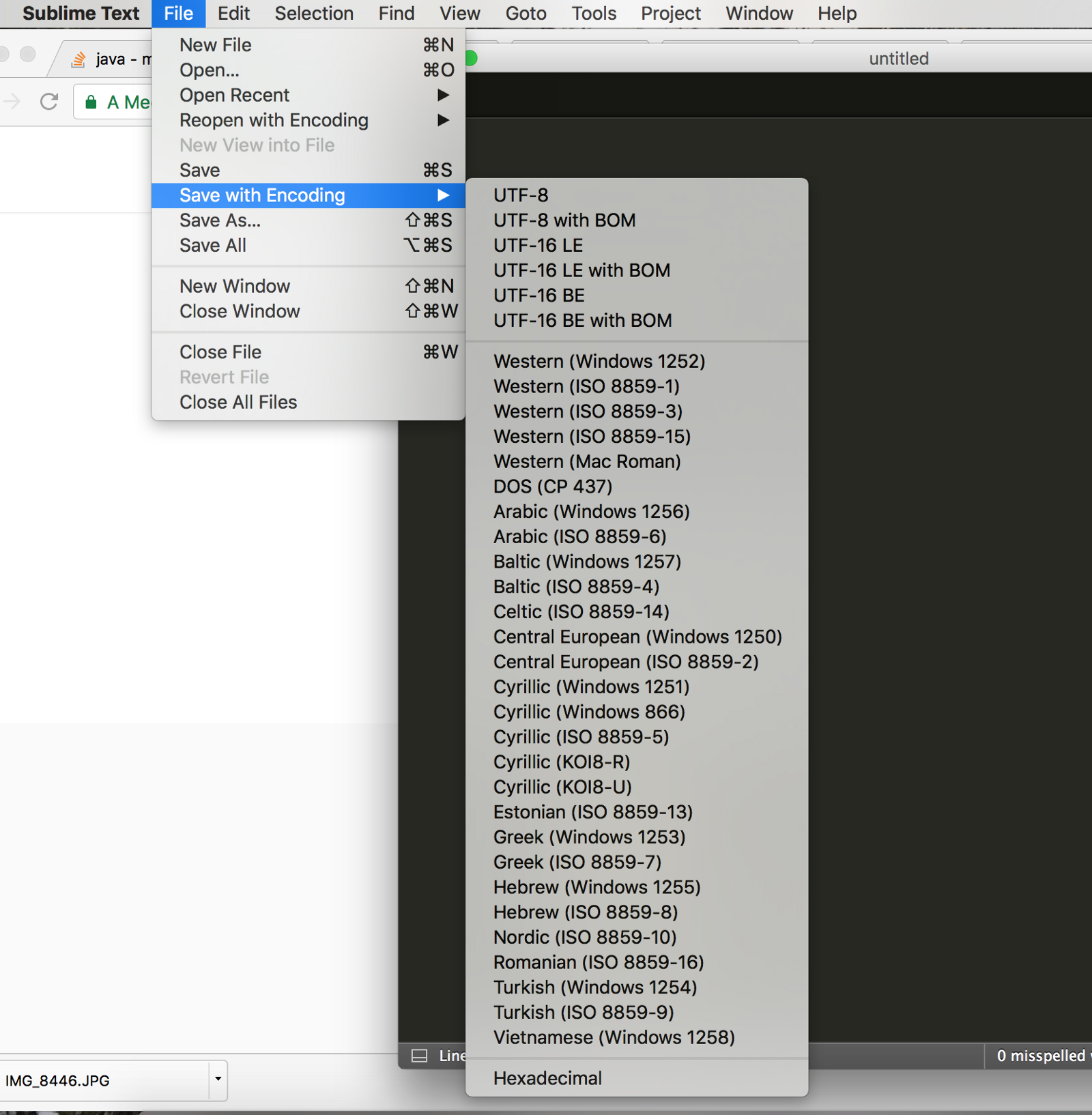
* Similar to the constant pool, each section starts with 2 bytes indicating its size, followed by a variable number of bytes defining the data
* Unlike the constant pool, the size bytes indicate the actual number of entries, not entries size – 1
* Each entry starts with a tag that indicates how much data is to come, and of what type
* For our Hello World program, we only care about Methods and we’ll leave the others blank – just 0000
* Now, we’re finally ready to code!

**Okay, Let’s Code!**

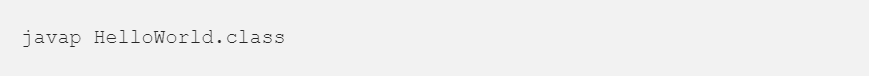
* If you didn’t already, download [**Sublime Text**](https://www.sublimetext.com/)
  + Works quite well for writing binary files from scratch in hex
    - allows you to save a file with hexadecimal encoding
  + Improves readability by allowing you to add white space between the Hex digits you are typing
* For our first class file, type this into your editor:



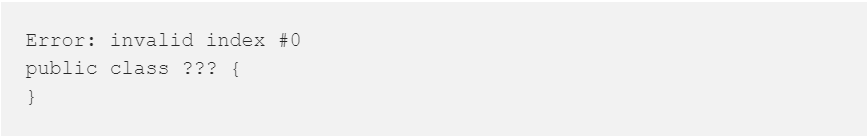
* This basically says it is a Java 8 Class file that is Super Public, invalid class indexes, with 0 interfaces, 0 fields, 0 methods, and 0 attributes
  + Java File: CAFE BABE
  + Version 8: 0000 0034
  + Constant Pool Size of ZERO: 0000
  + Super Public: 0021
  + Unknown index of class in constant pool: 0000
  + Unknown index of super class in constant pool: 0000
  + zero interfaces: 0000
  + zero fields: 0000
  + zero methods: 0000
  + zero attributes: 0000
* Save the file in **hexadecimal** encoding as HelloWorld.class



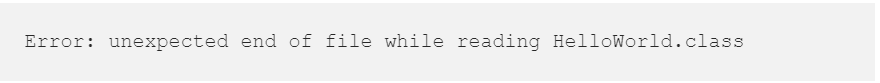
* To confirm it was done correctly, lets use the java class disassembler utility, javap
* At the command line, in the directory where you saved your file type:



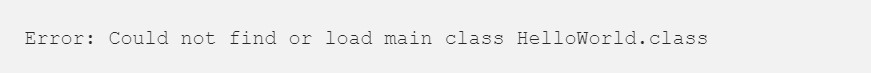
* If you wrote the file right, you’ll see this:



* + This is because we never put a class reference into our constant pool, and index 0 does not exist
  + This is just an empty unknown class
* Anytime you make a mistake you will most likely see this:



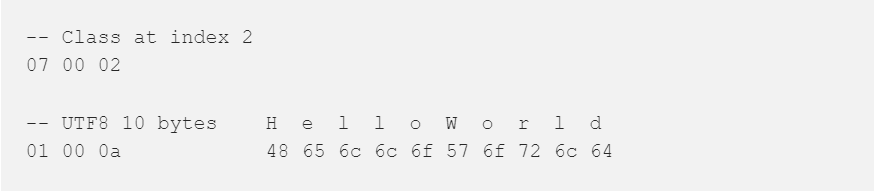
* If you run the file via *java HelloWorld.class*, you will see this output:



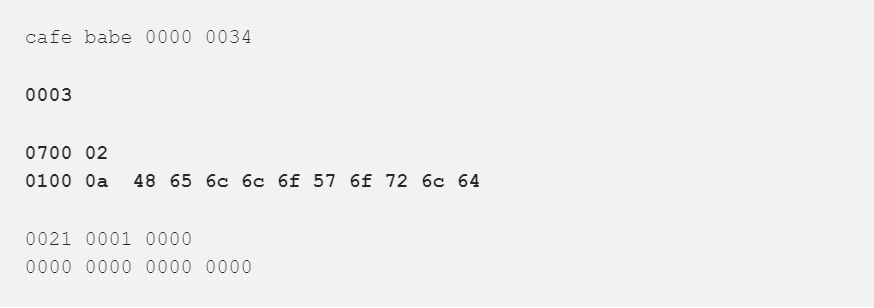
* + This error indicates that we have no main method

**Adding the HelloWorld Class name**

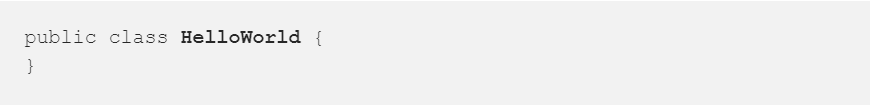
* Before we can add a main method, we must put our class in the constant pool and give it a name.
  + Class entries in the constant pool require two entries:
    - One to indicate that it is a class
    - Another for the UTF8 string data that is the class’ name
* The tag to create a class is **07**. The class constant pool entry is three bytes.
  + One byte for the tag and two bytes for an index pointing to a UTF8 entry in the constant pool.
* A UTF8 entry is denoted by the tag **01**. The tag is followed by two bytes that indicate the size in bytes of the UTF8 string.
  + That size is not the size of the string, but the number of bytes in the UTF8 string.
* The constant pool entries to add a class called “HelloWorld” would be:



* Let’s add those bytes to our file, and give our class a name:
  + Remember, Sublime Text will let you add space between the bytes to keep things somewhat readable

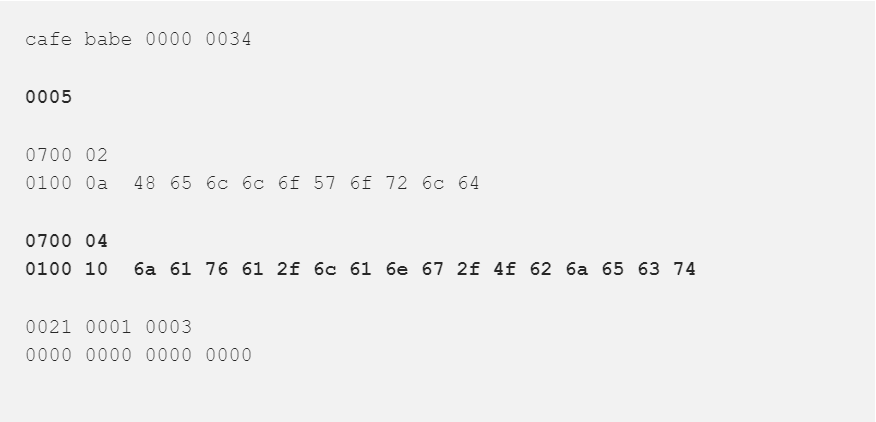


* Note that the size of our constant pool is **0003**. That is because the constant pool size is always 1 bigger than its actual size.
* Add that to your file and save it. Run *javap HelloWorld.class* again, and you should see:

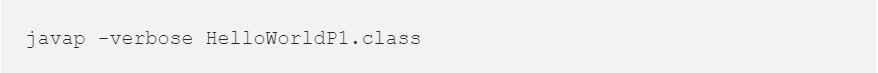


**Adding the Super Class**

* Now let’s add a super class for HelloWorld, and make it **java/lang/Object**.
  + use / instead of a period, unlike in Java source code
* Update your program to this:



* Here we have four entries in the constant pool (**0005**). 1 class definition pointing to the UTF8 value HelloWorld at index 2, and another class definition pointing to the UTF8 **java/lang/Object** at index 4.
  + After the access modifiers we specify which class it is, and what the index of its super is
* This time run:



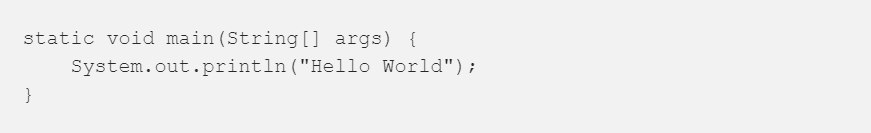
* This will show us more information so we can confirm our constant pool:



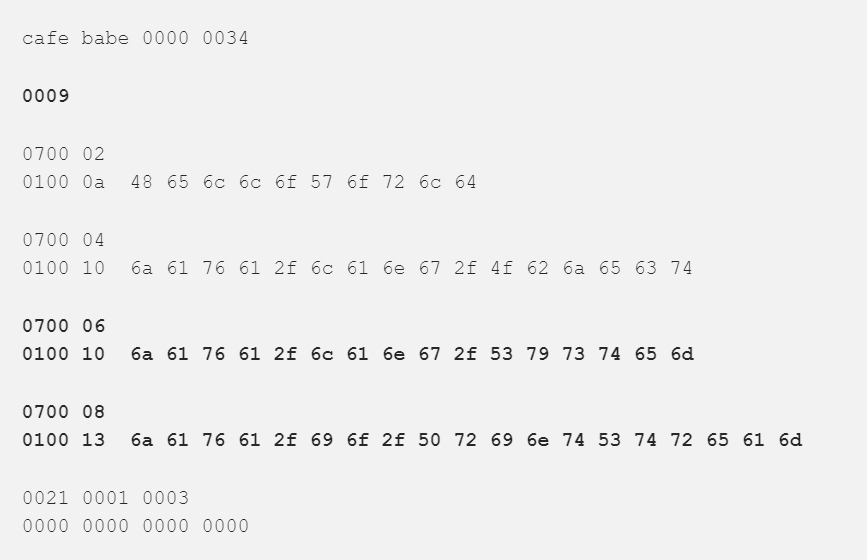
* Here we an see our version number of 52, our access modifiers, and our constant pool.
  + This is a great reference when hand coding binary!

**Filling out the Constant Pool**

* We are now ready to start work towards our Hello World main method!
* Our goal is this Java code:



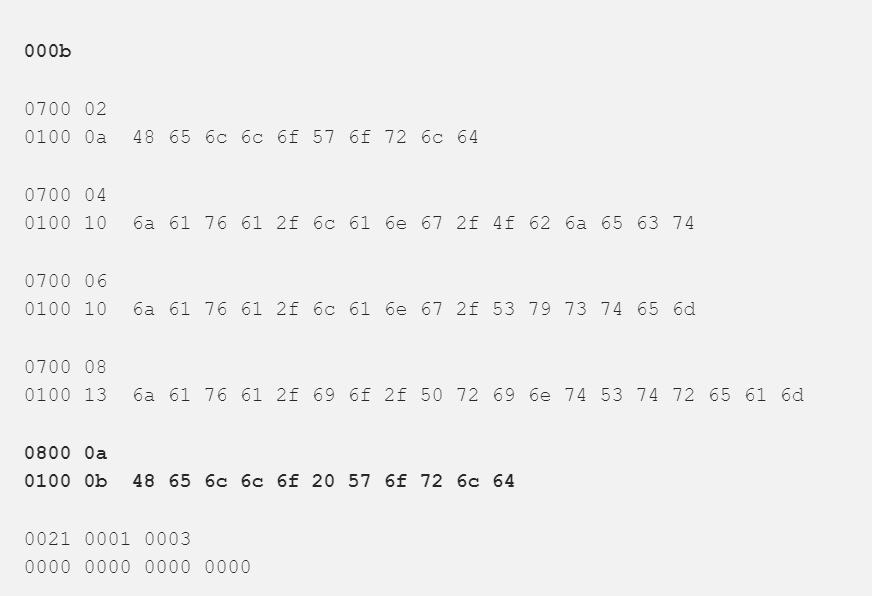
* Here is what we need in our constant pool in order to accomplish this:
  + One method reference to **println**, which in turn requires *two class references* to **System** and **PrintStream**, a variable reference to **out**
  + A constant string reference to **“Hello World”.**
  + UTF8 data for our method signature, 2 entries: **main** and **([Ljava/lang/String;)V**.
    - Those are used for the method name and return type.
  + A UTF8 string representing the special attribute **Code.**
    - This will be needed to indicate the body of the main method’s instructions
* First let’s add the two extra class references we need. **System** and **PrintStream**:



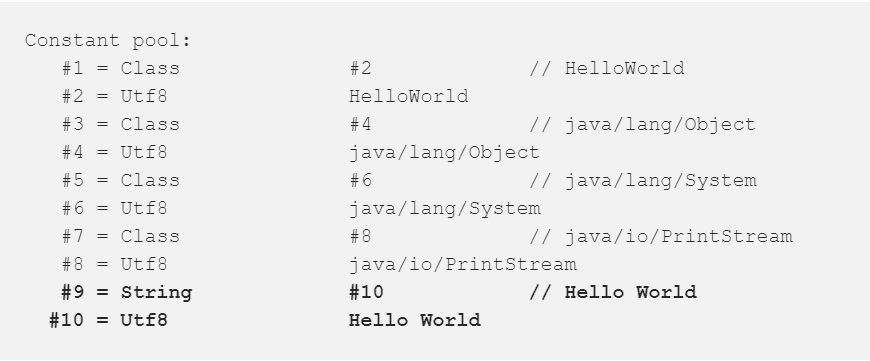
* Run **javap -verbose HelloWorld.class** and confirm you have a constant pool with these 8 entries:



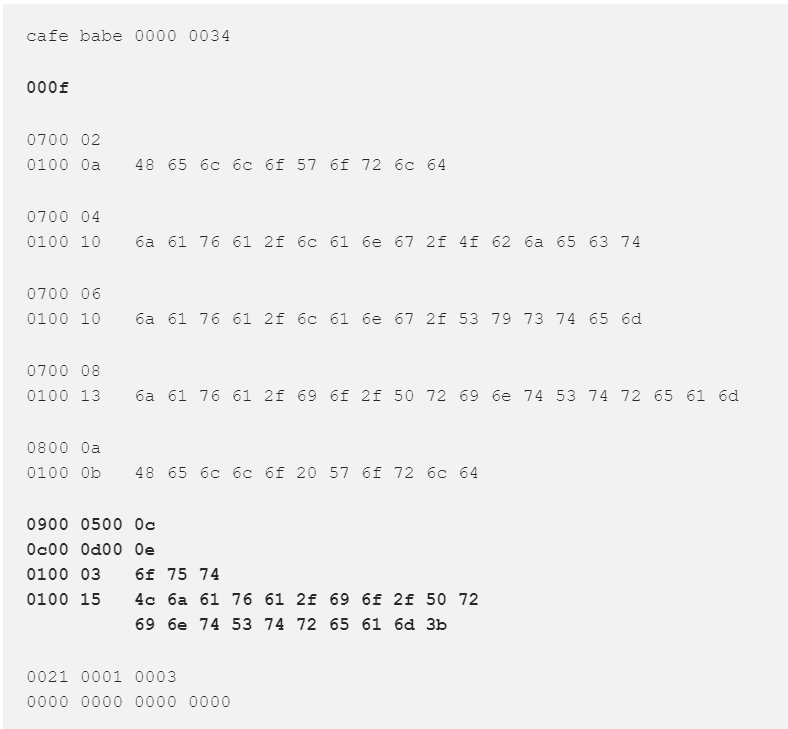
* Notice how the class entries have indexes pointing to the UTF8 entry that describes them. Make sure these line up!
* Next let’s add our one and only string constant we will be using in our program: **“Hello World”**
* Add a constant pool entry using the tag 08 to indicate a string constant. This entry is similar to the class entries, and should be 3 bytes long.
  + 1 byte for the tag
  + 2 bytes for the index of the UTF8 data of string
* Change your program to:



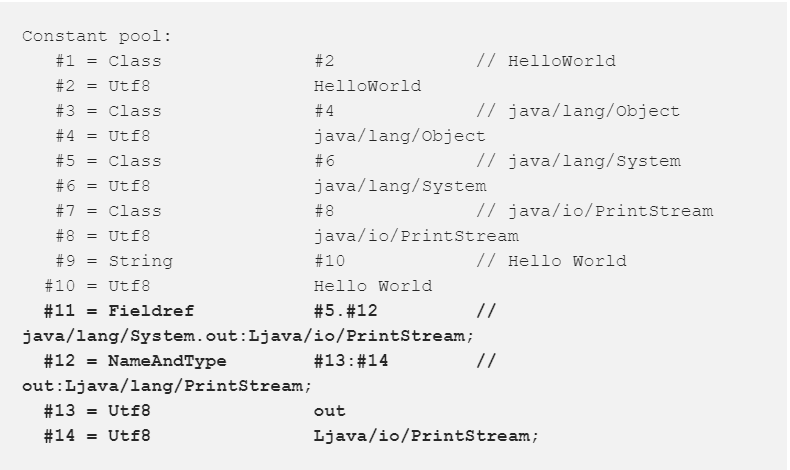
* Run **javap -verbose HelloWorld.class** and confirm your constant pool has two new entries.
  + These represent our **“Hello World”** string constant.



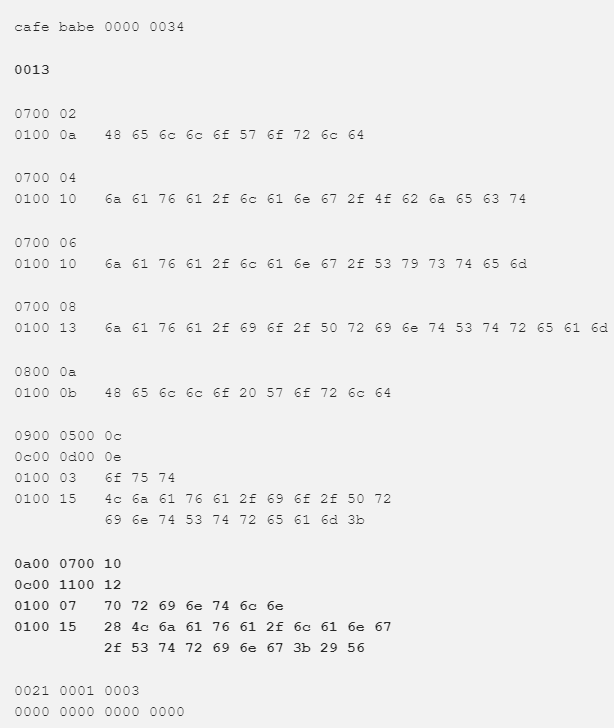
* Next let’s add our constant reference to the **out** static variable. Update your binary program with these bytes:



* **javap -verbose HelloWorld.class** should give you this:



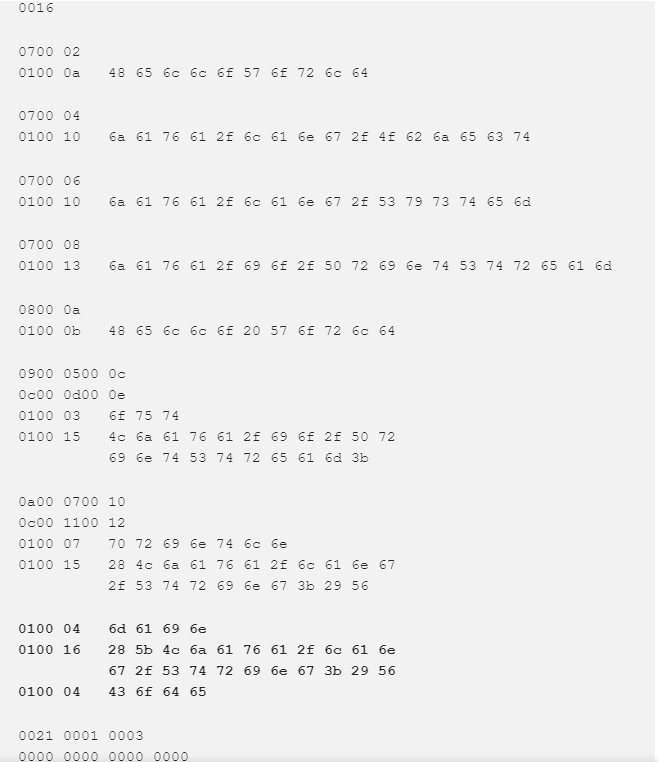
* We’ve added four new constant pool entries and introduced two new tag types here
  + Tag **09** is a field reference
  + Tag **0c** is a name & type reference
* A field reference is made up of 5 bytes
  + The first is the tag **09**
  + The next two bytes are the constant pool index to the class that the field belongs to
    - #7 **PrintStream** for us
  + The last two bytes point to a name and type reference for the field
  + All together the entry is the value **0900 0500 0c**
* The name & type reference for **out** we’ve added is: **0c00 0d00 0e**
  + **0c** is the tag type for NameAndType
  + **000d** is the name, a UTF8 entry at index #13, the value **out**
  + **000e** is the type, a UTF8 entry at index #14, the type of **Ljava/io/PrintStream;**
    - Note: Classes referenced as types always start with an **L.** We are also required to put a semicolon at the end of that string, because it could be followed by another type when defining method parameter types.
* Let’s move on and add our table entries for the **println** call. For these, we need a method reference in the constant pool:



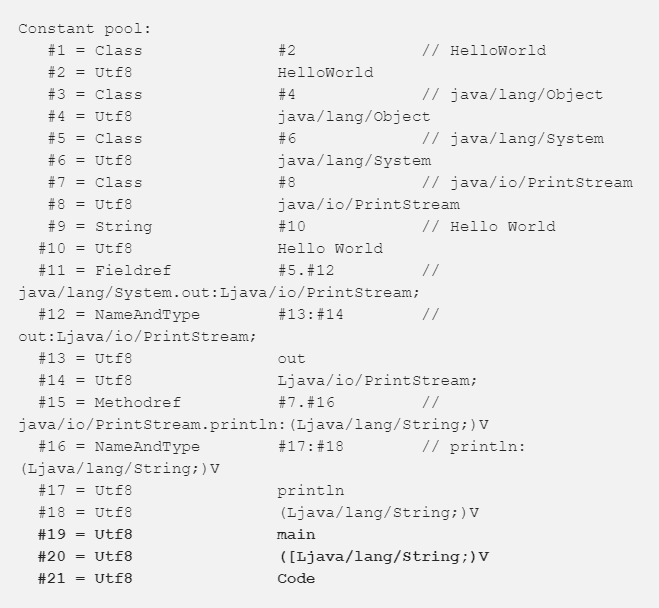
* **javap -verbose HelloWorld.class** gives us:



* We’ve added a method reference at index #15 of bytes: **0a00 0700 10**
  + The first byte is a tag, **0a**, which means method
  + The next 2 bytes indicates the constant pool index #7 class **PrintStream**
  + The final 2 bytes indicate the index of the **NameAndType** of this method
* The NameAndType we’ve added consists of 5 bytes: **0c00 1100 12**
  + **0c** is the tag type for NameAndType
  + **0011** indicates that the name of the method is at index #17
  + **0012** indicates that the type of the method is at index #18
    - A method type is written in the form: (Parameters)Return. In this case we take one String as a parameter, and return Void
      * **(Ljava/lang/String;)V** at index #18
* Before we finally write our main method instruction set, we need to create a few more UTF8 entries:
  + Two for our method, the name & type
  + Plus one more UTF8 entry for the special **Code** attribute
    - The **Code** attribute will indicate JVM machine instructions are coming
* Update your binary program to:

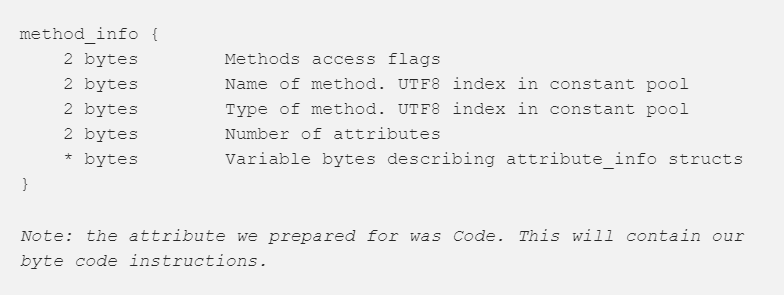


* **javap -verbose HelloWorld.class** gives us:

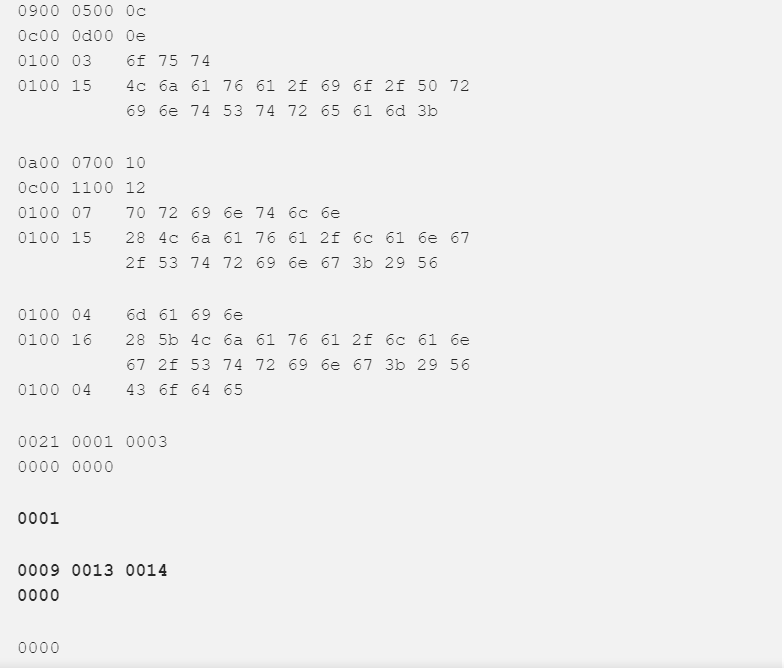


**Writing the HelloWorld main method**

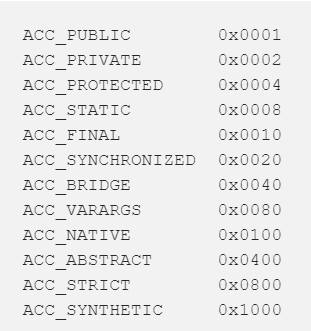
* We are finally ready to write our main method!
* Let’s add an empty static method to our byte code called main
* Method byte code follows this structure:



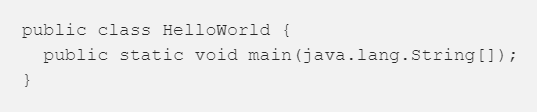
* Update your program to:



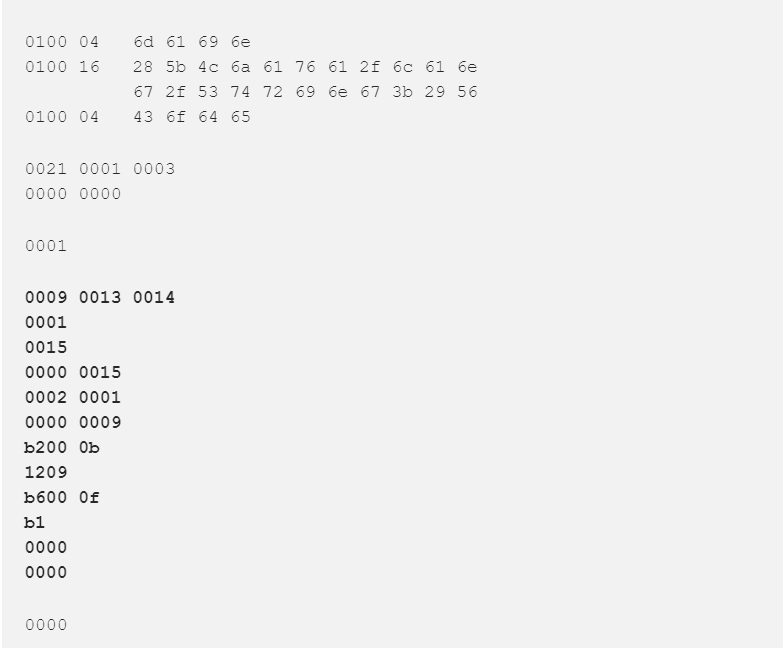
* This increases our number of methods to a count of 1 (**0001**) and adds the method signature.
* The first bytes **0009** are the access modifiers. Method access modifiers are as so:



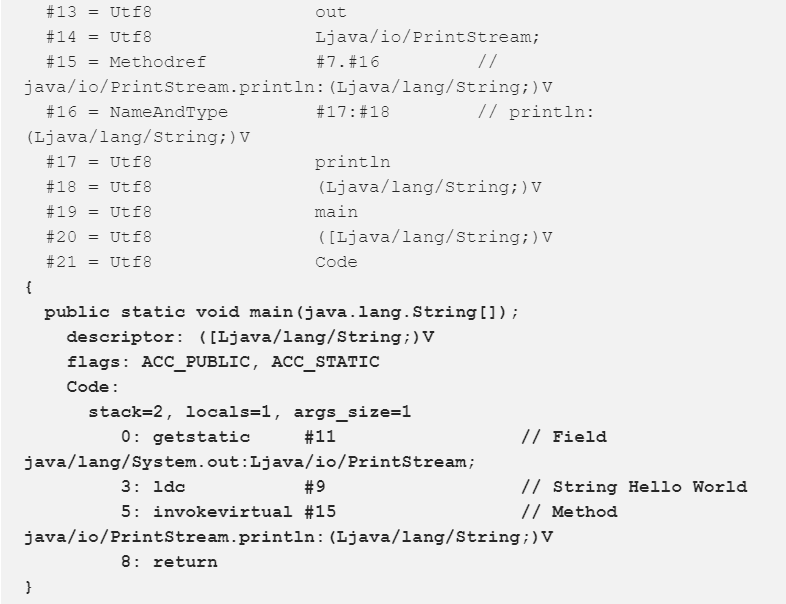
* **0009** indicates that the method we defined is public static
* The second 2 bytes **0013** are index #19 in our constant pool
  + This is the name of the method, main
* The next 2 bytes **0014** are index #20 in our constant pool.
  + **([Ljava/lang/String;)V**, the type of our method
  + They type says this method takes a primitive String array as a parameter and returns Void
* Running **javap HelloWorld.class** should give you:



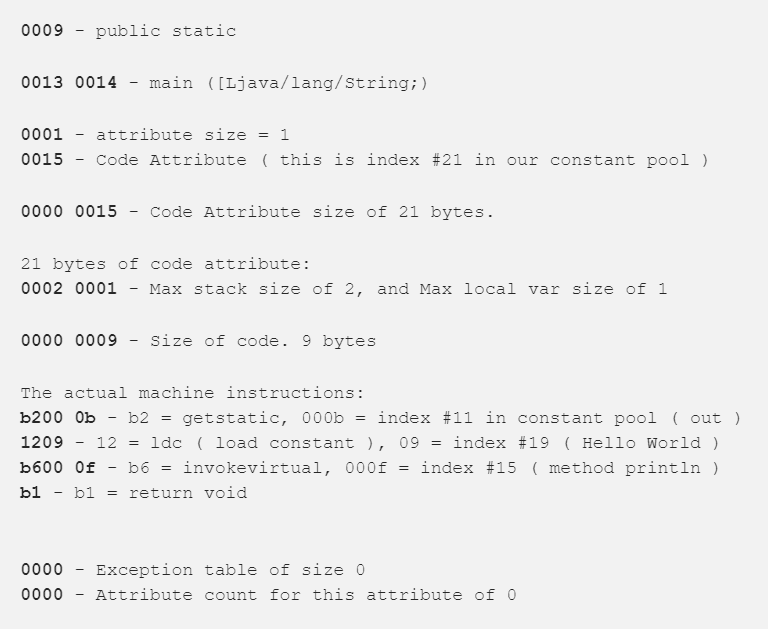
* We have our empty method stub! The last thing we need to do is to add our instructions for the method via a **Code** attribute.
* Update your program to:



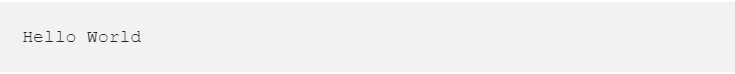
* **javap -verbose HelloWorld.class** should give you:



* Let’s examine all that byte code we added for our method:



* Now we can finally run our Hello World program with **java HelloWorld,** and it should give you this:



**No deliverable for this part!**